Micriμm

Empowering Embedded Systems

μC/USB-Bulk

and the
Philips LPC2148

Application Note

AN-7148-Bulk

www.Micrium.com

Table of Contents

1.00 Introduction 3
1.01 The Philips LPC2148 4
1.02 Device-Independent Files 5
1.03 Device-Dependent Files 6
1.04 IAR Embedded Workbench 7

2.00 Enumeration 8
2.01 Configuring your Device 8

3.00 USB Bulk IN Transfers 11
3.01 USB Bulk OUT Transfers 11

4.00 `app.c` 12

References 14

Contacts 14

1.00 Introduction

μC/USB-Bulk facilitates USB communication between your embedded device and a host PC according to the USB 1.1 or USB 2.0 specification. μC/USB-Bulk consists of ANSI C source code for your embedded processor, as well as a Windows device driver. Some of the features of μC/USB-Bulk are:

- A hardware abstraction layer that allows μC/USB-Bulk to be easily ported to different USB controllers.
- Support for bulk transfers, which can utilize a high percentage of bandwidth on the bus.
- Data rates as high as 12 Mbit/sec, which is the highest rate supported by the USB 1.1 specification.
- OS support. μC/USB-Bulk can be used with single-threaded applications, or with an RTOS, such as μC/OS-II.

1.01 The Philips LPC2148

This document provides guidelines for developing µC/USB-Bulk-based applications for the Philips LPC2148 microcontroller. The LPC2148, which features an ARM7TDMI-S CPU, includes a USB device controller that complies with the USB 2.0 specification. Figure 1-1 shows the LPC2148 on Keil's MCB2140 Evaluation Board, the platform that was used to test the files included with this document. These files were developed using IAR's Embedded Workbench, but other tool chains can be used to develop applications based on the files.

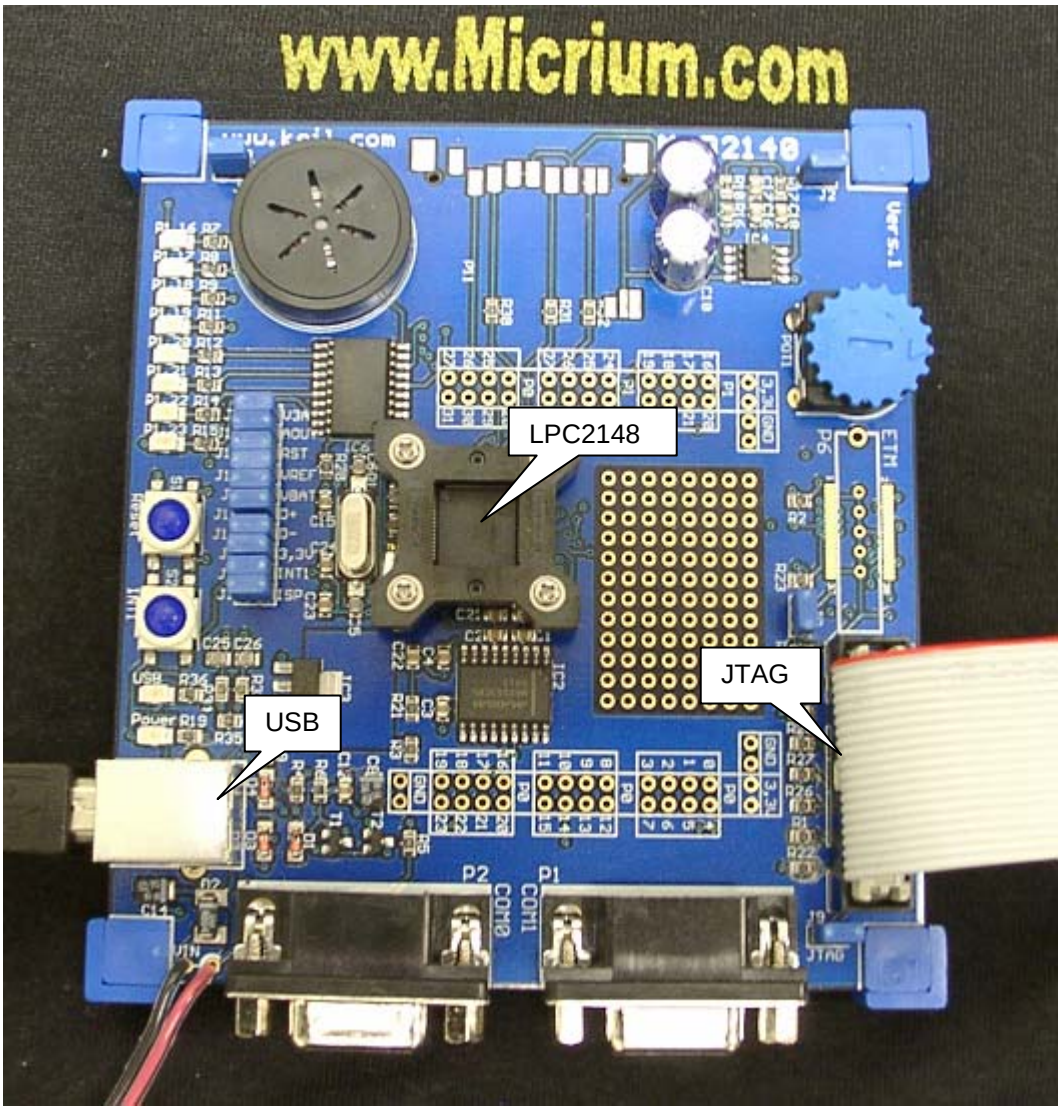

Figure 1-1, The Keil MCB2140 Evaluation Board
http://www.keil.com/mcb2140/

1.02 Device-Independent Files

µC/USB-Bulk, like Micrium's other software modules, is provided in source code form, and it is compiled along with your application. µC/USB-Bulk's source code consists of a combination of device-independent files and files that adapt the software to specific USB controllers. AN-7148-Bulk.zip, the file that includes this document, only furnishes the hardware-specific µC/USB-Bulk files. If you wish to build the example project included in AN-7148-Bulk.zip, you will need to acquire the device-independent source files for µC/USB-Bulk, as well as those for µC/OS-II, from Micrium. After you have obtained these files, you should place them in the folders described below.

\Micrium\Software\uCOS-II\Source

The processor-independent code for µC/OS-II should be placed in this folder. Version 2.81 or newer should be used.

\Micrium\Software\uC-USB\Firmware\USBBulk

µC/USB-Bulk's device-independent files should be located in this folder. These files, which are listed below, declare the generic read and write routines that are called by your application.

```
USB.h
USB_Private.h
USB_Main.c
USB_Read.c
USB_Setup.c
USB_Write.c
```

\Micrium\Software\uC-USB\Firmware\Config

µC/USB-Bulk's configuration files should be placed in this folder. These files define configuration constants, such as the vendor and product IDs, that allow you to customize the appearance of your device to the host. The following configuration files are needed:

```
USB_Conf.h
USB_Descriptors.c
```

\Micrium\Software\uC-USB\Firmware\System\OS\uCOS-II

This directory contains USB_X_uCOS-II.c, which creates the µC/OS-II objects used by µC/USB-Bulk.

\Micrium\Software\uC-USB\Driver

The Windows device driver for µC/USB-Bulk should be located in this directory.

\Micrium\Software\uC-USB\SampleApp

This directory holds a sample application, SampleApp.exe, for your PC.

1.03 Device-Dependent Files

In addition to the device-independent files listed in the previous section, the example project that accompanies this document requires several LPC2148-specific files. These files are included in `AN-7148-Bulk.zip`, and in order to build the example project, the files must be located in the folders described below.

`\Micrium\Software\uCOS-II\Ports\ARM`

This directory contains the µC/OS-II ARM ports.

`\Micrium\Software\uCOS-II\Ports\ARM\Generic`

This directory holds 'generic' ARM ports (i.e. ports that can be used with any target board).

`\Micrium\Software\uCOS-II\Ports\ARM\Generic\IAR`

This directory contains the processor-specific µC/OS-II files listed below. These port files, which are compatible with processors running in ARM or Thumb mode, are thoroughly described in Application Note AN-1014. This document can be downloaded from the Micrium Web site.

```
os_cpu.h
os_cpu_a.s
os_cpu_c.c
os_dbg_c.c
```

`\Micrium\Software\uC-USB\Firmware\System\Device\LPC2148_USB`

This directory holds the hardware-specific µC/USB-Bulk files for the LPC2148 USB device. The two files listed below serve as an interface between the µC/USB-Bulk device-independent code and the LPC2148 USB controller.

```
USB_LPC2148_reg.h
USB_hw.c
```

`\Micrium\Software\EvalBoards\Philips\LPC2148-Keil\IAR\OS-USB`

This folder contains the LPC2148 source code corresponding to the PC-based sample application listed in the previous section. This directory contains the following files:

```
app.c
app_cfg.h
includes.h
os_cfg.h
LPC2148-OS-USB.*
```

`app.c` is the actual application, and `includes.h` is a master include file used by the application. `os_cfg.h` and `app_cfg.h` are configuration files for µC/OS-II and the application, respectively. `LPC2148-OS-USB.*` denotes the IAR Embedded Workbench project files.

`\Micrium\Software\EvalBoards\Philips\LPC2148-Keil\IAR\BSP`

The Board Support Package for the Keil MCB2140 board resides in this folder.

1.04 IAR Embedded Workbench

The IAR Embedded Workbench (EW) V4.30a was used to develop the files included with this document. You can of course use µC/USB-Bulk and µC/OS-II with other tools. Figure 1-2 shows the project tree for the files included with this document.

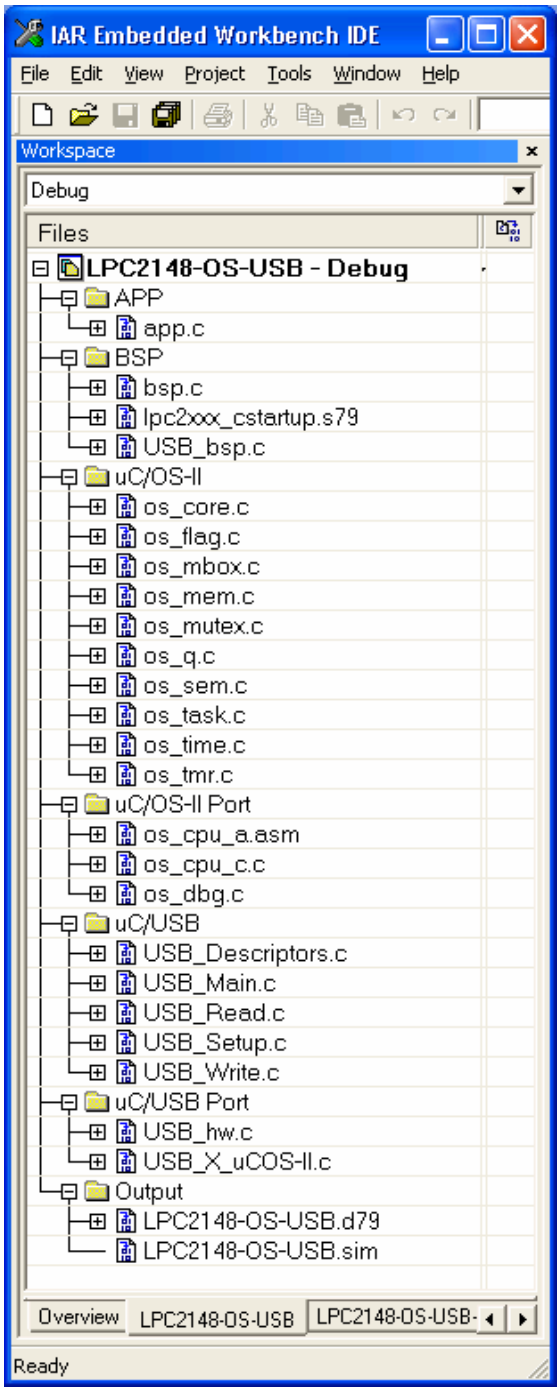

Figure 1-2, IAR EW Project

2.00 Enumeration

Once you have populated the folders described in sections 1.02 and 1.03, you can use the IAR Embedded Workbench to build `LPC2148-OS-USB.ewp`, the project included with this document. You should be able to build this project with no errors or warnings, and you can run the resulting executable file using the IAR C-SPY debugger. If the application is running correctly, you will notice that one of the MCB2140 board's LEDs rapidly blinks on and off.

The board's blinking LED indicates that the device is waiting to be enumerated. Enumeration is the means by which a USB host identifies USB devices, and this process is normally performed when a device is first connected to a host. Since the MCB2140 board is normally connected to the host as a means of providing power to the board, the host most likely attempted enumeration before your application was loaded, and this unwanted enumeration could preclude your application from receiving the requests that it expects from Windows. For this reason, the example project utilizes IAR's flash loader for the LPC2148, allowing you to download the project to Flash memory, and ensuring that an application capable of enumeration begins executing whenever the board is connected to a host.

Although this automatic enumeration is desired in a completed USB product, it can cause problems when you are debugging your device. If, upon connecting the board to a host, the application loaded from flash even partially completes enumeration, Windows may not attempt to enumerate the device again when a new application is loaded with C-SPY, leaving you unable to properly debug the latter application. To prevent this situation, you will either need to load the Flash memory with a dummy application that does not enable the USB device, or switch to an external power source that will allow you to connect to the host after the application that you intend to debug has been loaded.

Once you have chosen a strategy to prevent a previously loaded application from hindering your debugging process, you should make sure that your application performs the initializations that will allow it to be enumerated. Mainly, you should ensure that your application invokes `USB_Init()`, an initialization function that is called by the provided sample application. `USB_Init()` is responsible for setting up the hardware accessed by μC/USB-Bulk, and it should be called before your application uses any of the μC/USB-Bulk API functions. Your application also needs to invoke `USB_X_Init()`, which initializes the operating system objects needed by μC/USB-Bulk, before `USB_Init()` is called. After performing these initializations, your application will be ready to use μC/USB-Bulk's API functions to send and receive data. The included sample application indicates the completion of these initializations by ceasing to toggle the LED that blinks during enumeration.

2.01 Configuring your Device

μC/USB-Bulk provides two configuration files, `USB_Conf.h` and `USB_Descriptors.c`, that allow you to make changes to the information that is provided to the host PC during enumeration. This information includes the vendor and product IDs, which are represented by the constants `USB_Vendor_ID` and `USB_Product_ID` in `USB_Conf.h`. The vendor and product IDs are also defined in `USBBulk.inf`, which is provided with μC/USB-Bulk and resides on the host PC. If these ID numbers need to be changed, for example, after a vendor ID is obtained from the USB Implementers Forum, the appropriate constants must be changed in both files. The μC/USB-Bulk User's Manual contains further information about modifying `USB_Conf.h` and `USBBulk.inf`.

Along with the vendor and product IDs, the USB host uses the descriptors that it obtains from the device during enumeration to select an appropriate driver. μC/USB-Bulk defines these descriptors in USB_Descriptors.c. This file defines a device descriptor, USB_acDeviceDescriptor[], which contains general information about the device, as well as a configuration descriptor, USB_acConfigDescriptor[], which includes information about the device's endpoints. USB_Descriptors.c also contains four string descriptors, which are referenced by the other descriptors.

The descriptors defined in USB_Descriptors.c may need to be edited to allow the hardware-specific μC/USB-Bulk files to function properly. You should make sure that the declarations in USB_Descriptors.c match the descriptors listed below, since these descriptors were used to test the application included with this document. The device descriptor is shown in Listing 2-1, and USB_acConfigDescriptor[], which defines the configuration descriptor as well as its subordinate interface and endpoint descriptors, can be seen in Listing 2-2.

Listing 2-1, USB_acDeviceDescriptor[]

```
const  unsigned  char  USB_acDeviceDescriptor[0x12] = {
    sizeof(USB_acDeviceDescriptor),    /* Descriptor length          */
    USB_DESC_TYPE_DEVICE,              /* Descriptor type            */
    USB_STORE_U16(0x110),              /* USB specification number   */
    0x00,                              /* Device class               */
    0x00,                              /* Device subclass            */
    0x00,                              /* Device protocol            */
    0x40,                              /*            (1)             */
    USB_STORE_U16(USB_VENDOR_ID),      /*            (2)             */
    USB_STORE_U16(USB_PRODUCT_ID),
    USB_STORE_U16(1),                  /* Device release number      */
    STRING_INDEX_MANUFACTURER          /*            (3)             */
    STRING_INDEX_PRODUCT,
    STRING_INDEX_SN,
    0x01                               /* Number of configurations   */
};
```

L2-1(1) This field of the descriptor indicates the maximum packet size for Endpoint 0, the endpoint that is used for the control transfers that facilitate enumeration. A maximum packet size of 64 bytes is specified, and this value should not be changed unless corresponding changes are made to the hardware-specific routines that access Endpoint 0.

L2-1(2) Through these two fields of the descriptor, the host PC can obtain the product and vendor IDs defined in USB_Conf.h.

L2-1(3) These three fields are used to specify a manufacturer name, product name, and serial number for the device. These fields do not contain the actual values; they hold references to the string descriptors containing these values. The string descriptors are defined in USB_Descriptors.c. Like the vendor and product IDs, if these descriptors are changed, USBBulk.inf must be modified to reflect the changes.

Listing 2-2, USB_acConfigDescriptor[]

```
const  unsigned  char  USB_acConfigDescriptor[9+9+7+7] = {
    9,                                                  /* Descriptor length          */
    USB_DESC_TYPE_CONFIG,                               /* Descriptor type            */
    USB_STORE_U16(sizeof(USB_acConfigDescriptor)),      /*           (1)              */
    1,                                                  /* Number of interfaces       */
    1,                                                  /*           (2)              */
    0,                                                  /* Configuration string index */
    0x80,                                               /*           (3)              */
    50

                                                        /* Interface descriptor       */

    0x09,                                               /* Descriptor length          */
    USB_DESC_TYPE_INTERFACE,                            /* Descriptor type            */
    0x00,                                               /* Interface number           */
    0x00,                                               /* Alternate settings         */
    0x02,                                               /*           (4)              */
    0xFF,                                               /* Interface class            */
    0xFF,                                               /* Interface subclass         */
    0xFF,                                               /* Interface protocol         */
    0,                                                  /* Interface string index     */

                                                        /* Endpoint 2 IN Descriptor   */

    0x07,                                               /* Descriptor length          */
    USB_DESC_TYPE_ENDPOINT,                             /* Descriptor type            */
    0x82,                                               /*           (5)              */
    USB_TRANSFER_TYPE_BULK,
    USB_STORE_U16(64),
    0,                                                  /* Polling interval           */

                                                        /* Endpoint 2 OUT Descriptor  */

    0x07,                                               /* Descriptor length          */
    USB_DESC_TYPE_ENDPOINT,                             /* Descriptor type            */
    0x02,                                               /*           (6)              */
    USB_TRANSFER_TYPE_BULK,
    USB_STORE_U16(64),
    0,                                                  /* Polling interval           */
};
```

L2-2(1) The total length of the configuration descriptor and its subordinate descriptors is given by this field.

L2-2(2) This field assigns a numerical value to the configuration set forth by the descriptor. A USB host can use this value to configure the device via a `Set_Configuration` request.

L2-2(3) This field informs the USB host that the device will be powered by its USB connection. The maximum amount of current that the device will require is specified in the subsequent field.

L2-2(4) The total number of endpoints used, not counting Endpoint 0, is denoted in this field.

L2-2(5) This field indicates that this endpoint descriptor pertains to Endpoint 2 IN. The next two fields note that this endpoint will be used for bulk transfers with a maximum packet size of 64 bytes.

L2-2(6) This field identifies Endpoint 2 OUT as the subject of this endpoint descriptor. Like Endpoint 2 IN, Endpoint 2 OUT will handle bulk transfers with a maximum packet size of 64 bytes.

3.00 USB Bulk IN Transfers

Your application can send data to the USB host by calling `USB_Write()`. This function sends data using Endpoint 2, and it requires two parameters: a pointer to the buffer holding the data that will be sent, and the amount of data to send. For example, to send the first 10 bytes of an array called `Test_Array` to the host, the following call would be used:

```
USB_Write(Test_Array, 10);
```

Since all USB transfers are initiated by the USB host, data sent using `USB_Write()` will not actually be transferred to the host until one or more IN packets are received. In fact, because Endpoint 2 only has 128 bytes of buffer space, calls to `USB_Write()` with more than 128 bytes will not return until enough data has been transferred to allow the remaining bytes to be placed in the buffer. There is basically no limit on how much data can be transferred through a call to `USB_Write()`, though, provided that all of the data will ultimately be requested by the USB host. The constant `MAXT_IN`, which is defined in `USB_Private.h`, and should have a value of `64` for the LPC2148, does not indicate the maximum amount of data that can be transferred with `USB_Write()`. This constant instead reflects the maximum number of bytes that are transferred by `USB_HW_SendData()`, the hardware-specific function that `USB_Write()` uses to transfer data.

3.01 USB Bulk OUT Transfers

`USB_Read()` can be called by your application to read data that has been received from the host. This function is analogous to `USB_Write()`; it takes a pointer to the buffer that will hold the received data, along with a size parameter. For example, to read 10 bytes from the host into an array called `Test_Array`, the following call would be used:

```
USB_Read(Test_Array, 10);
```

As with `USB_Write()`, if a call to `USB_Read()` is issued and no data has been received from the USB host, the function will wait for the reception of packets. If, conversely, packets are received before a call to `USB_Read()` has been made, these packets will still be available when the application is prepared to read them. Whether or not packets are received before the call to `USB_Read()`, however, the function will not return until all of the requested bytes are received.

4.00 app.c

app.c, which is included with this document, is a simple application that you can use to verify that μC/USB-Bulk is functioning properly. After running the application using the C-SPY debugger, you should see one of the MCB2140 board's LEDs blink, indicating that enumeration is taking place. After enumeration is complete and the LED has stopped blinking, the application's only task, which is shown in Listing 4-1, below, calls USB_Read() to receive a single byte from the host. At this point, you can run SampleApp.exe, the sample application for your PC. This application will send the requested byte and then wait for the device to send the byte back, incremented by one. This send and receive process is repeated, and after it has finished, you should see the message displayed in Figure 4-1.

Listing 4-1, app.c

```
static  void  AppTaskStart (void *p_arg)
{
    INT8U  c;


    (void)p_arg;

    BSP_Init();                             /*                (1)                   */

#if OS_TASK_STAT_EN > 0
    OSStatInit();                           /* Initialize the uC/OS-II statistics task  */
#endif

    USB_X_Init();                           /*                (2)                   */
    USB_Init();                             /* Initialize the USB hardware             */

    while (USB_IsConfigured() == 0) {       /* Wait for the device to become configured */
        USB_X_Delay(50);
        LED_Toggle(1);
    }

    LED_On(2);

    while (1) {                             /*                (3)                   */
        USB_Read(&c, 1);                    /* Receive one byte of data from the host   */

        c++;                                /* Increment the received byte             */

        USB_Write(&c, 1);                   /* Send one byte of data to the host        */
}
```

L4-1(1) BSP_Init() initializes the peripherals used by μC/OS-II. These peripherals include a Timer/Counter, which serves as μC/OS-II's tick source, and the Vectored Interrupt Controller. BSP_Init() also initializes the LPC2148's PLL to create a system clock running at 48 MHz.

L4-1(2) As noted in Section 2.00, Enumeration, USB_X_Init() and USB_Init() must be called before any μC/USB-Bulk API functions are used.

L4-1(3) A call to USB_Read() is made with the intention of receiving a single character. Once this character has been received, it is incremented and sent back to the host. The sample application on the PC should be used to send and receive characters anytime after enumeration has completed.

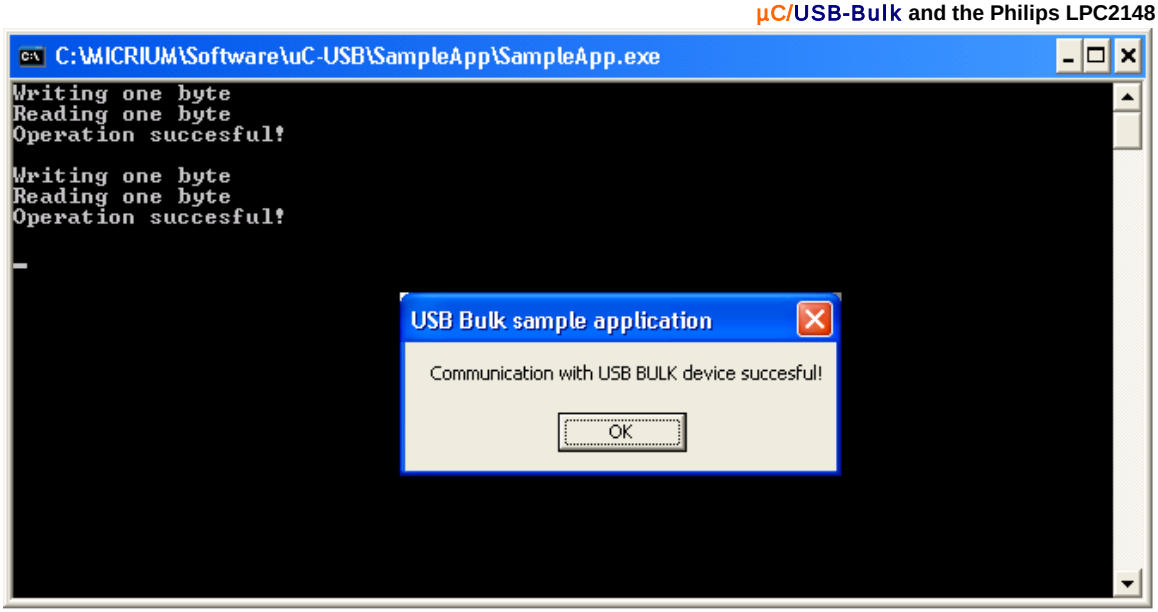

Figure 4-1, Successful USB Communication

References

μC/OS-II, The Real-Time Kernel, 2nd Edition
Jean J. Labrosse
R&D Technical Books, 2002
ISBN 1-57820-103-9

Embedded Systems Building Blocks
Jean J. Labrosse
R&D Technical Books, 2000
ISBN 0-87930-604-1

Contacts

IAR Systems
Century Plaza
1065 E. Hillsdale Blvd
Foster City, CA 94404
USA
+1 650 287 4250
+1 650 287 4253 (FAX)
e-mail: Info@IAR.com
WEB : www.IAR.com

R&D Books, Inc.
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-9950
USA
+1 785 841 1631
+1 785 841 2624 (FAX)
e-mail: rushorders@cmpbooks.com
WEB: http://www.cmpbooks.com

Micriμm
949 Crestview Circle
Weston, FL 33327
USA
+1 954 217 2036
+1 954 217 2037 (FAX)
e-mail: Jean.Labrosse@Micrium.com
WEB: www.Micrium.com